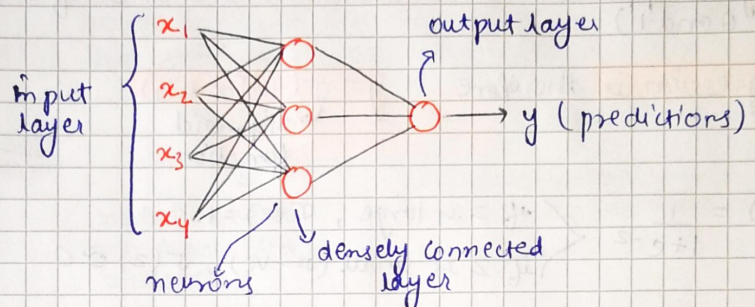
Coursera: **Deep Learning specialization**

Neural networks and deep learning

① **Neuron** → Within an artificial neural network, a "neuron" is a mathematical function that models the functioning of a biological neuron.
→ Typically, a neuron compute the weighted average of its input, and this sum is passed through a non-linear function, often called "activation function".



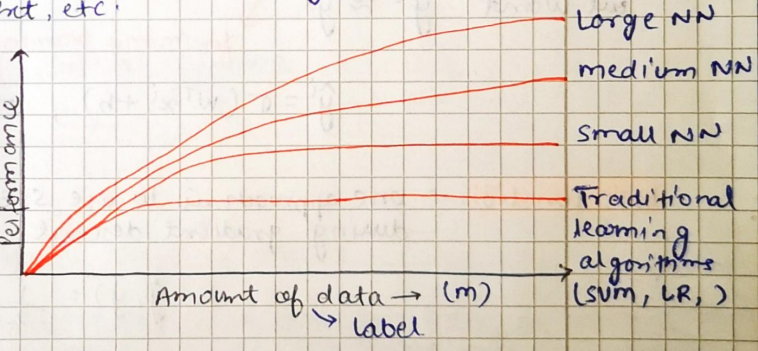
② **Supervised Learning** → Also known as supervised machine learning, is a subcategory of machine learning and AI. It is defined by its use of labelled datasets to train algorithms to classify data or predict outcomes accurately.

Structured data → Structured data, in terms of deep learning means that every feature has a well-defined meaning.

Unstructured data → Unstructured data can be categorised in Audio signals, image, text, etc.

③ Deep learning progress →

→ Deep learning is progressing by scaling. Scaling meaning with the size of the network as well as size of the data



④ **Logistic Regression** → It can be defined as an algorithm used for binary classification

Binary classification → If we have an RGB image, we make an input vector having the dimensions $(n \times n \times 3)$ where n = Eg. 64×64, 128×128, etc.

Notation ① Single training example → (x, y) → $y \in \{0, 1\}$ → labels

② Training set = $\{(x^1, y^1), (x^2, y^2) \dots (x^m, y^m)\}$ → $x \in R^{n_x}$ } input vector

m_{train} → no. of training examples, m_{test} → no. of test examples

③ $X = \begin{bmatrix} | & | & & | \\ x^1 & x^2 & \dots & x^m \\ | & | & & | \end{bmatrix}$ (n_x rows, m columns)

$X \in R^{n_x \times m}$ → matrix with the dimension $n_x \times m$

④ $Y = [y^1, y^2, \dots, y^m]$ $\{y \in R^{1 \times m}\}$

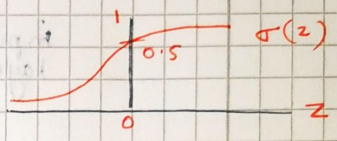
→ Given an input vector x (image), we want to predict whether it is a cat or not or

$\hat{y} = P(y=1|x)$ → label "1" given the value of $x \in R^n$

The parameters which we use are $w \in R^{n_x}$, $b \in R$

One idea is to use linear regression (i.e. $\hat{y} = w^T x + b$). But it seems it is not a very good approach since we want a chance of our output (i.e. a value b/w 0 and 1)

Logistic Regression is therefore $\hat{y} = \sigma(w^T x + b)$ (z) → sigmoid function



$\sigma(z) = \frac{1}{1+e^{-z}}$ if z is large, $\sigma(z) = 1$; if z is small (or -ve), $\sigma(z) \approx 0$

* when implementing logistic Regression, our goal should be to learn the parameters w and b so that $\hat{y}$ becomes a good estimate of y being equal to 1

→ Logistic Regression cost function → To train the parameters "w" and "b" of our model, we define a cost function.

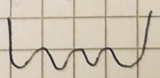
given $\{\hat{y} = \sigma(w^T x + b), \ \sigma(z) = \frac{1}{1+e^{-z}}\}$ and $\{(x^1, y^1), (x^2, y^2) \cdots (x^m, y^m)\}$

we want $\hat{y}^i \approx y^i$ → for many training examples,

$\hat{y}^i = \sigma(w^T x^i + b)$, $\sigma(z^i) = \frac{1}{1+e^{-z^i}}$

Loss function → one approach is to use SSE, but since it does not works good during gradient descent, we have to find another way.

$L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$



A good approach is to use: $L(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$

an approximation of this is that

① if $y=1$: $L(\hat{y}, y) = -\log \hat{y}$ ⇒ for $y=1$, we want $-\log \hat{y}$ to be as small as possible meaning $\hat{y}$ to be very big. But since we know that $\hat{y}$ is a function of sigmoid, it can never be greater than 1.

② if $y=0$: $L(\hat{y}, y) = -\log(1-\hat{y})$ ⇒ Similarly, we want $\log(1-\hat{y})$ to be large, and therefore we want $\hat{y}$ to be as small as possible.

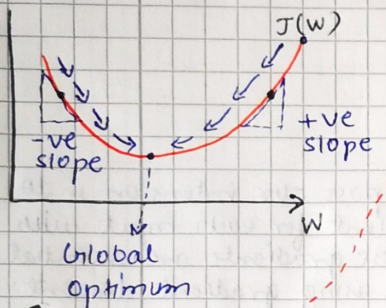
Cost function : $J(w, b) = \frac{1}{m} \sum_{i=1}^{m} L(\hat{y}^i, y^i)$ → over the entire training set.

→ **Gradient Descent** → It is an optimization algorithm which is used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient.

In machine learning, we use gradient descent to update the parameters (w, b) of our model. Parameters refers to the "coefficients" in linear regression and weights in neural networks.

note ★ In other words, we want to find w and b that minimizes $J(w,b)$

Let's say we have a cost function $J(w,b)$ which we want to minimize. What gradient descent does is, it repeatedly performs the following updating steps:



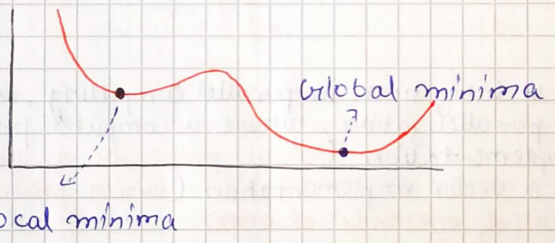
$$w := w - \alpha \frac{dJ(w)}{dw}$$ → Update

- $:=$ notation which represents the update
- α → learning rate which denotes how big a step would take in each iteration

Until the algorithm converges at Global optimum.

★ Global optima → A global minimum of a function is a point where the function value is smaller than all other feasible points.

Local optima → A local minimum of a function is a point where the function value is smaller than at nearby points, but possibly greater than at a distant point.



→ why this update rule makes sense is that:

① If we are on the rising side (+ve slope) of the function, we would have a +ve slope, therefore $w = w - (\cdots)$ i.e. the value will be reduced.

② If we are on the falling side (-ve slope) of the function, we would have a -ve slope, therefore $w = w + (\cdots)$ i.e. the value will be increased.

⇒ The final update rule becomes

$$w := w - \alpha \frac{dJ(w,b)}{dw} \longrightarrow \frac{\partial J(w,b)}{\partial w}$$

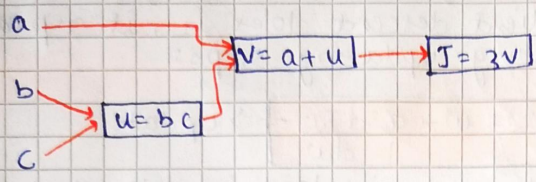
$$b := b - \alpha \frac{dJ(w,b)}{db} \longrightarrow \frac{\partial J(w,b)}{\partial b}$$

In case of multivariable we use $\partial(del)$ instead of $d(d)$

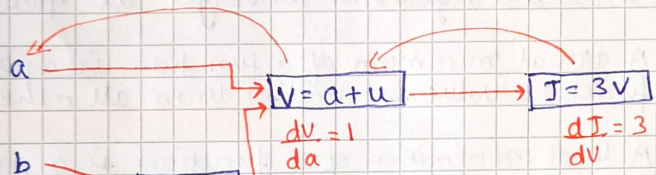
→ **Computation graph** → It is defined as a directed graph where the nodes corresponds to mathematical operations. Computational graphs are a way of expressing and evaluating a mathematical expression.

Eg. $J(a,b,c) = 3(a+bc)$

$u = bc$
$v = a+u$
$J = 3v$



Backward Pass (Back propogation) → In the backword pass, our intension is to calculate the gradient for each input with respect to the final output. These gradients are essential for training the neural network using gradient descent.



$\frac{dJ}{dv} = 3$, $\frac{dJ}{da} = 3$

$\frac{dv}{da} = 1$

$\frac{dJ}{da} = \frac{dJ}{dv} \cdot \frac{dv}{da}$
$= 3 \cdot 1$
$= 3$

$\frac{dJ}{db} = \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{db}$

$\frac{dJ}{dc} = \frac{dJ}{dv} \cdot \frac{dv}{du} \cdot \frac{du}{dc}$

$\frac{dv}{db} = c$, $\frac{du}{dc} = b$

→ **Vectorization** → Automatic vectorization, in parallel computing, is a special case of automatic parallelization, where a computer program is converted from a scalar implementation, (which processes a single pair of operands at a time) to a vector implementation (which processes one operation on multiple pair of operands at once).

* whenever possible avoid "for-loops". Instead use in-built functions.

→ Calculating the derivatives of logistic Regression (with and with Vectorization)

```
J=0, dw1=0, dw2=0, db=0
for i=1 to m:
    z^i = w^T x^i + b
    a^i = σ(z^i)
    J += -[y^i log ŷ^i + (1-y^i) log (1-ŷ^i)]

    dz^i = a^i (1-a^i)
    dw1 += x1^i dz^i
    dw2 += x2^i dz^i
    db += dz^i

J = J/m, dw1 = dw1/m, dw2 = dw2/m
db = db/m
```

```
J=0, dw = np.zeros((nx,1)), db=0
for i=1 to m:
    z^i = w^T x^i + b
    a^i = σ(z^i)
    J += ....

    dz^i = a^i (1-a^i)
    dw += x^i dz^i

    db += dz^i

J = J/m, dw /= m, db = db/m
```

Broadcasting → The term broadcasting refers to how numpy treats arrays with different dimensions during arithmetic operations which leads to certain constraints, the smaller array is broadcast across the larger array so that they have compatible shapes.